Integrating monoliths with message brokers

Michał Cebula, Software Engineer @ MAXIO

Agenda

- Story of the two monoliths - what problem do we want to solve?
- Possible tooling - RabbitMQ
- Commons and differences between message brokers
- Transactional Outbox Pattern
- Implementation details - Shuttle SDK
- Current state - where are we now and where are we heading to?

MAXIO

The story of the two monoliths

Chargify and SaaSOptics are coming together as a one company - MAXIO

Current situation – API sync

Issues with REST API communication

Current situation – API sync

Issues with REST API communication

- Synchronously only
- Dependency on availability - no reliability
- No persistence
- Response time
- Inefficient scaling
- No communication flow management

Possible tooling?

What is RabbitMQ?

MAXIO

What is RabbitMQ?

Open-source message queuing system for asynchronous communication between systems in a distributed environment. It is based on Pub/Sub and Message Queue patterns. It allows sending, receiving, storing and managing message flow in an efficient and reliable way.

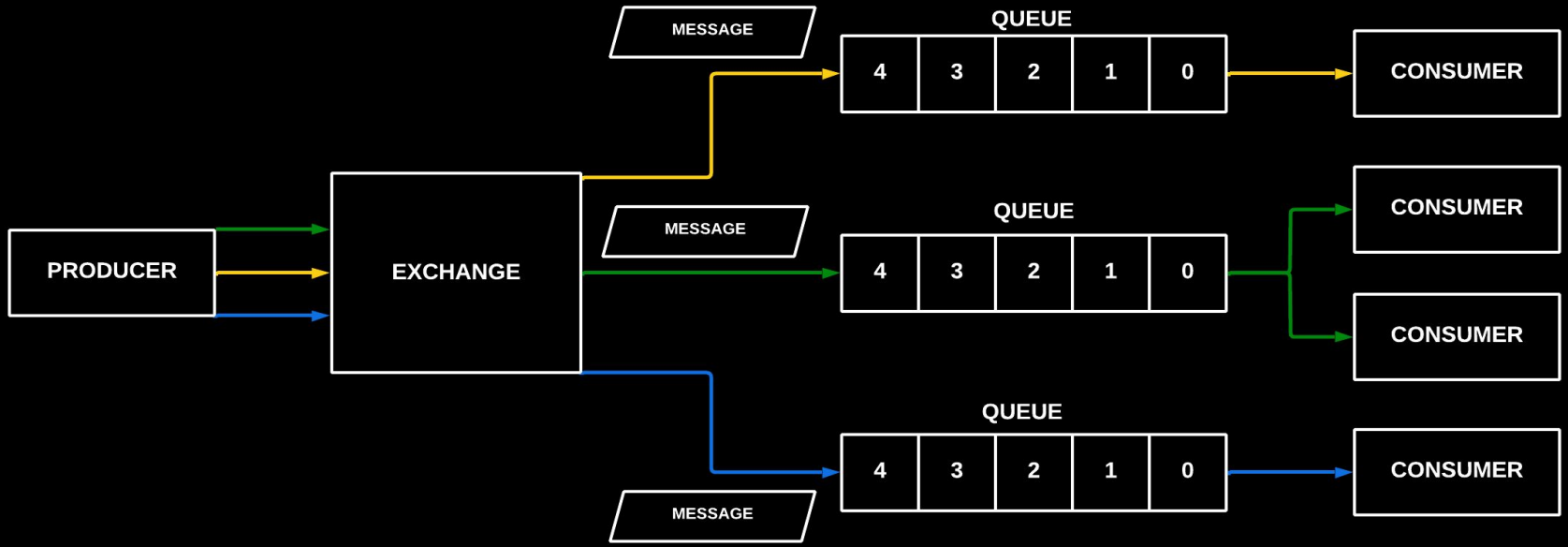
It was developed in 2007 and is written in Erlang. It implements the AMQP (Advanced Message Queue Protocol) model.

RabbitMQ – Properties

- Works as a server
- Provides CLI for managing queues – *rabbitmqclt*
- There are two kinds of rabbitMQ: queues (AMQP) and stream (similar to Kafka)
- From 4.0 replicated data structures were moved to Quorum Queues and Streams

MAXIO

RabbitMQ – Design



MAXIO

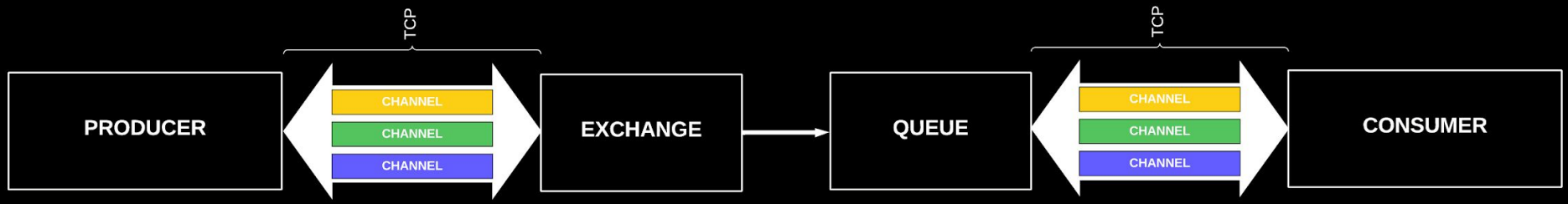
RabbitMQ

- Asynchronous communication
- Cross-platform
- Scalability
- Publish-Subscribe model
- Queuing
- Acknowledgment
- Persistence of messages
- Failures management and retries
- Transactions
- Security

MAXIO

RabbitMQ - Channels

Through channels producer or consumer is connected with the exchange. Channels are a logical connections within a single TCP connection. That allows the systems to not keep many connections open and saves the resources.



MAXIO

RabbitMQ – Message

- Routing key
- Body
- Properties

MAXIO

RabbitMQ – Producer

```ruby
require 'bunny'

connection = Bunny.new(automatically_recover: false)
connection.start

channel = connection.create_channel
queue = channel.queue('hello')

channel.default_exchange.publish('Hello World!', routing_key: queue.name)
puts " [x] Sent 'Hello World!'"

connection.close
```

MAXIO

RabbitMQ – Consumer

```ruby
require 'bunny'

connection = Bunny.new(automatically_recover: false)
connection.start

channel = connection.create_channel
queue = channel.queue('hello')

begin
  puts ' [*] Waiting for messages. To exit press CTRL+C'
  # block: true is only used to keep the main thread
  # alive. Please avoid using it in real world applications.
  queue.subscribe(block: true) do |_delivery_info, _properties, body|
    puts " [x] Received #{body}"
  end
rescue Interrupt => _
  connection.close

  exit(0)
end
```

MAXIO

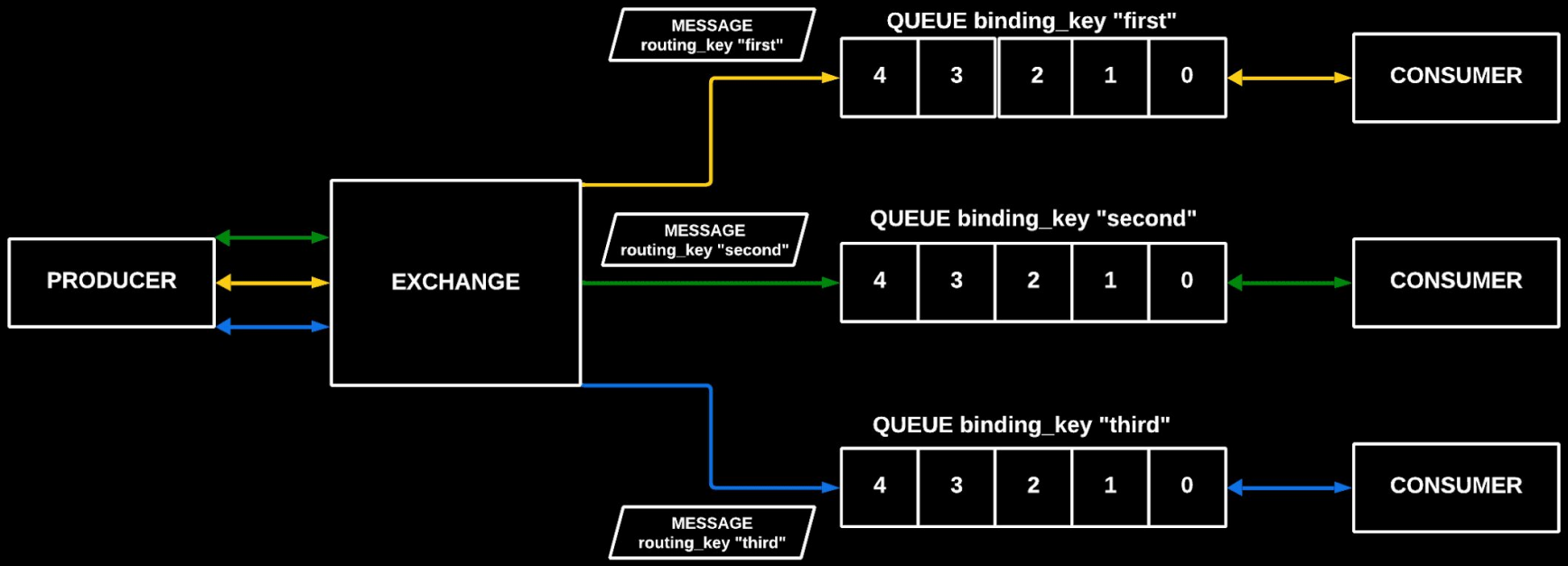
RabbitMQ - Acknowledgements

RabbitMQ is supporting manual and automatic acknowledgments.

MAXIO

RabbitMQ - Communication Models

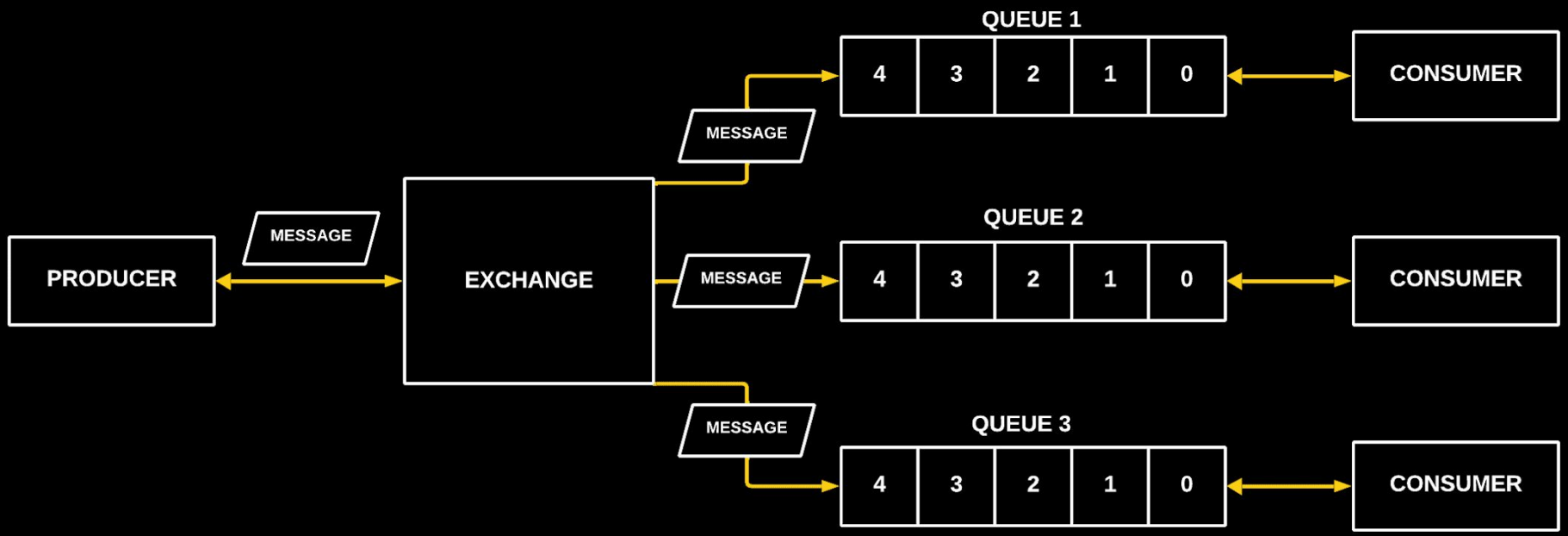
Direct Exchange



MAXIO

RabbitMQ – Communication Models

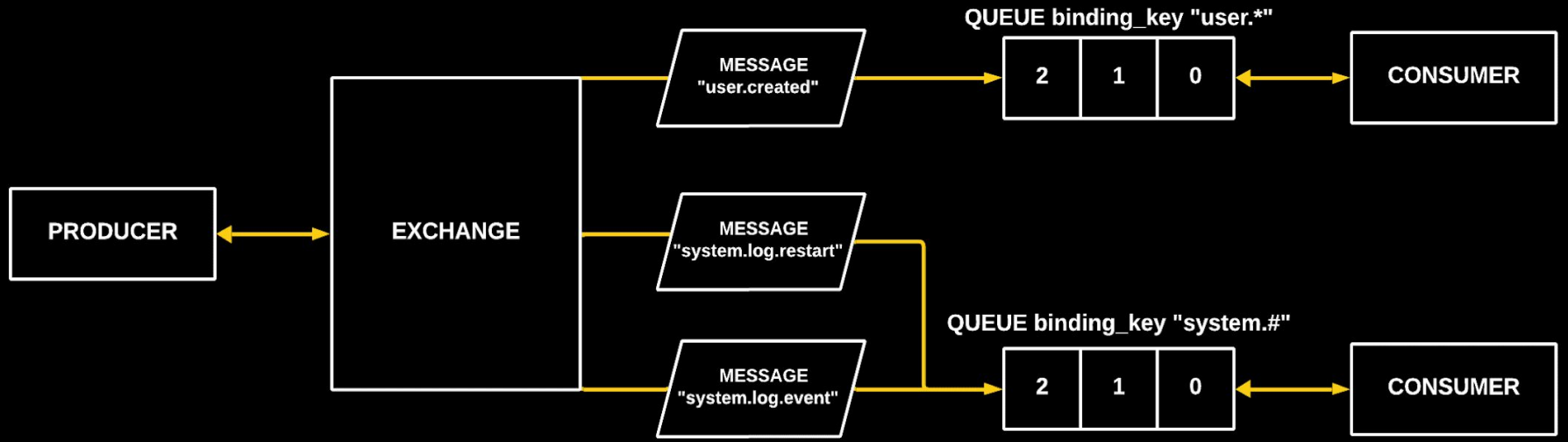
Fanout Exchange



MAXIO

RabbitMQ – Communication Models

Topic Exchange



MAXIO

RabbitMQ vs. Apache Kafka

RabbitMQ

- Moderate data volumes
- Complex and direct routing
- Handles more data within a message
- Messages deleted after they are consumed
- Messages are usually intended for a single consumer
- Used for decoupling systems

Kafka

- Very high throughput
- Fan out by default
- Handles simple and uniformed messages - events
- Messages are stored according to persistence policy
- Messages are usually broadcasted to multiple consumers
- Used for streaming data

MAXIO

Transactional Outbox Pattern

Transactional Outbox Pattern

What is the problem?

Transactional Outbox Pattern

What is the problem?

Our existing monoliths have persistence layers and we want to avoid scenarios in which published messages/events are inconsistent with the state of the relational database.

Image situations when:

Transactional Outbox Pattern

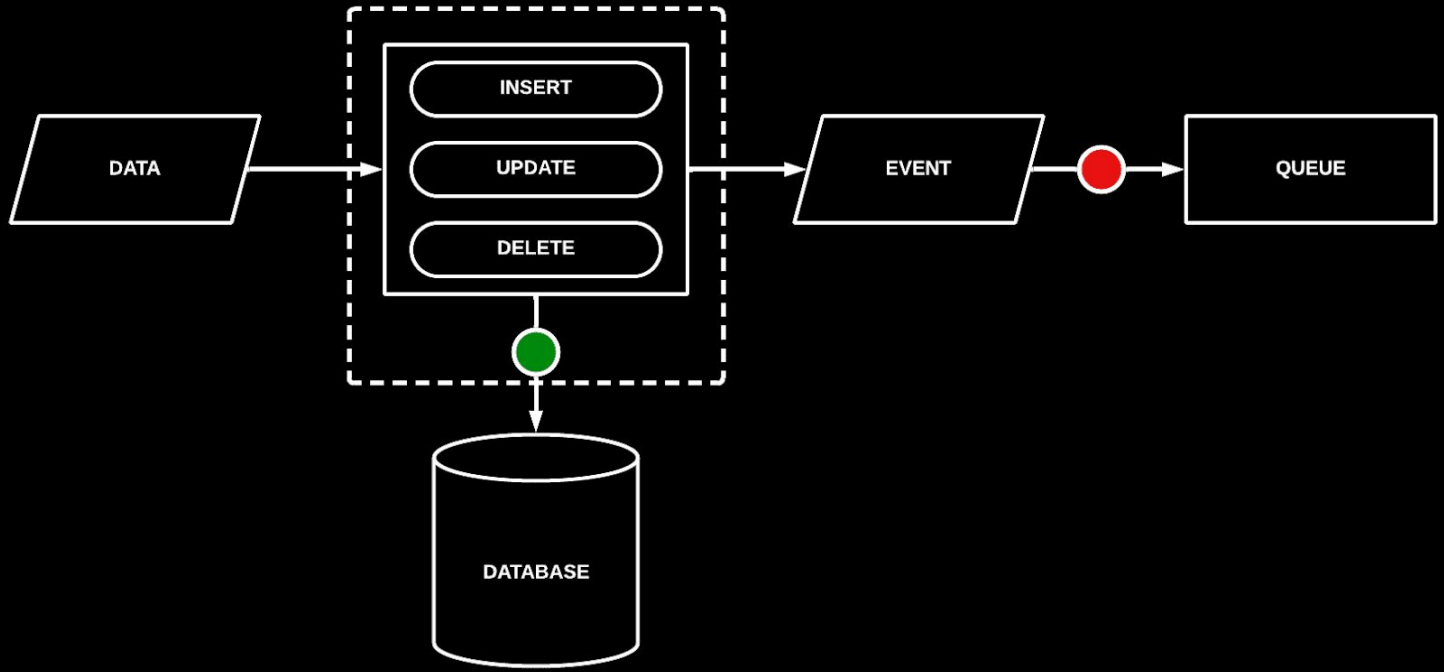
What is the problem?

Our existing monoliths have persistence layers and we want to avoid scenarios in which published messages/events are inconsistent with the state of the relational database.

Image situations when:

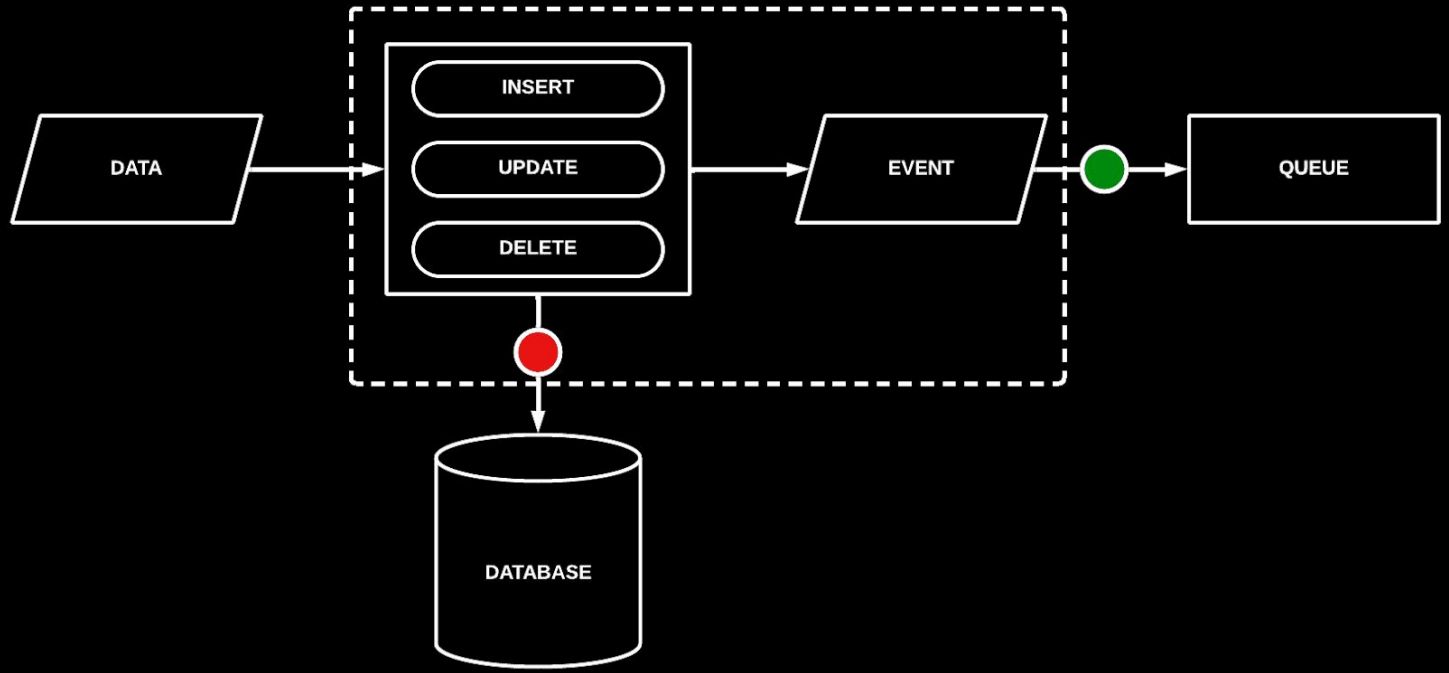
- message being sent within a database transaction before that transaction was rolled back.
- transaction succeeded, but afterward something happened and event has not been emitted.
- race condition in which the transaction eventually succeeds, but the emitted event is processed before the persistence layer reflects the consistent state.

Transactional Outbox Pattern



MAXIO

Transactional Outbox Pattern



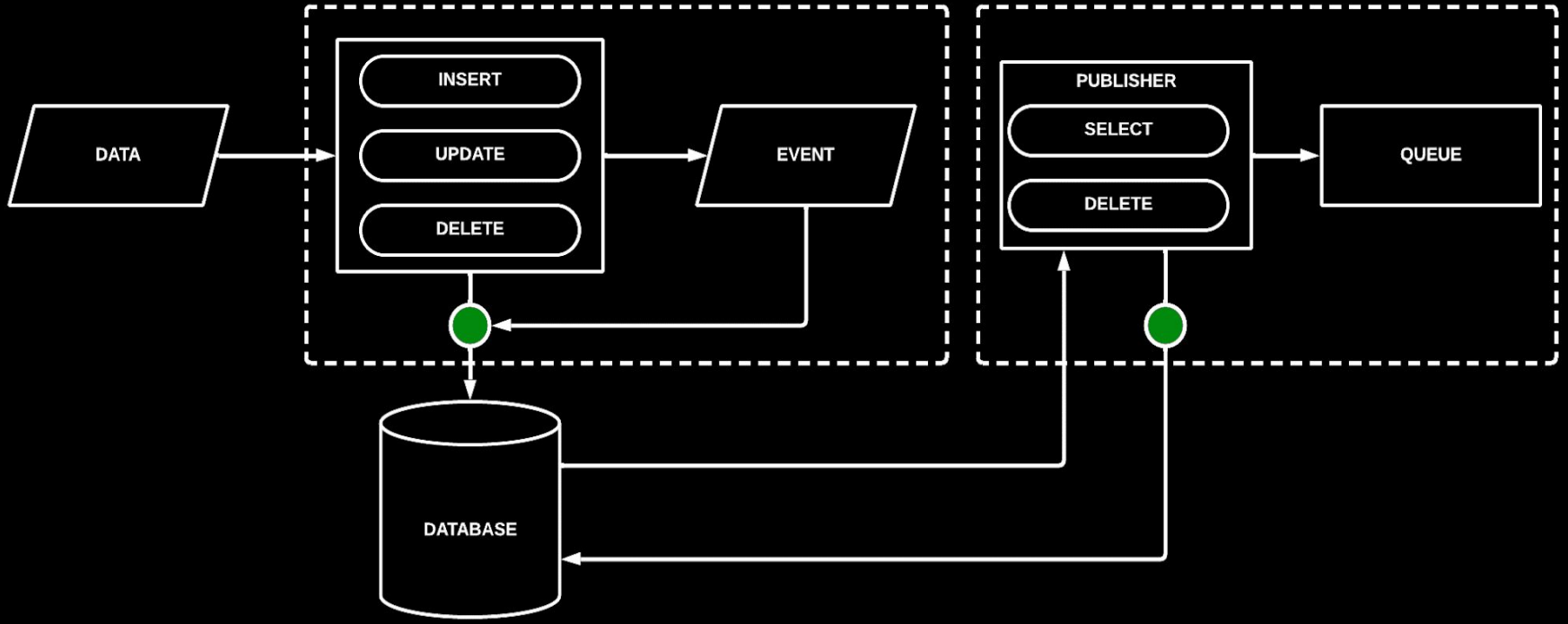
MAXIO

Transactional Outbox Pattern

How to solve it?

Instead of directly emitting an event within or after the transaction, we can create a new Outbox database table that stores all the properties required to send a proper message to our broker. Then the separate process (job, scheduler etc.) reads the record and handles events in a desired order.

Transactional Outbox Pattern



MAXIO

Transactional Outbox Pattern

What do you need to do?

- Create a new *Outbox* table.
- Persist a record within the transaction whenever your data will change.
- Run the proces to read from the *Outbox* table and emit and event to the broker.
- After the event is successfully emitted, delete the record from the *Outbox* table.

MAXIO

Transactional Outbox Pattern

What are the main benefits?

- Data consistency
- Message persistence

MAXIO

Transactional Outbox Pattern

Other benefits

- Filtering messages
- Records from the outbox table can stand as a log
- Inbox pattern

Transactional Outbox Pattern

Possible problems?

- It does not prevent from sending duplicate messages.
- Increased processing time.
- If publisher service is running on multiple instances, locking on reading records from outbox table may be required.
- Unsend message can block sending other messages.

Implementation details – Shuttle SDK

MAXIO

Implementation details – Shuttle SDK

SHUTTLE SCHEMA

SHUTTLE RELAY

SHUTTLE CONSUMER

MAXIO

Shuttle Schema

- Provides information about all events that can be emitted
- Uses Avro as data serialization system
- Defines the event schemas - keys and types
- Defines errors structure
- Provides tools for AR objects serialization and deserialization

MAXIO

Shuttle Relay

- Defines and implements Producer logic and configuration
- Implements Transactional Outbox Pattern
- Provides tools for publishing events without additional overhead
- Handles errors and retries

MAXIO

Shuttle Consumer

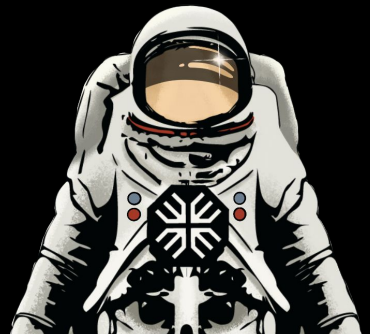
- Provides the tool for generating event-specific consumers
- Allows to configure retries strategy, logger or errors handler

MAXIO

Current state

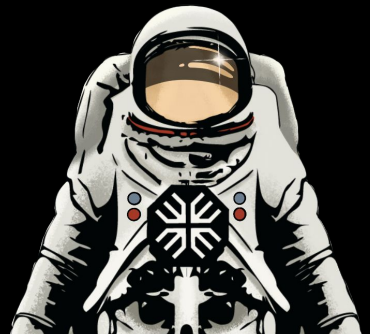
- There is not a single source of truth
- Normalization issues - same entities existing in both systems and schemas are not unified
- All exchanges and queues are durable
- Each entity type has its own queue
- Multiple consumers subscribed to a single queue
- Retries are handled with a dead-letter queue
- Errors are handled by emitting a new event

Dziękuję!



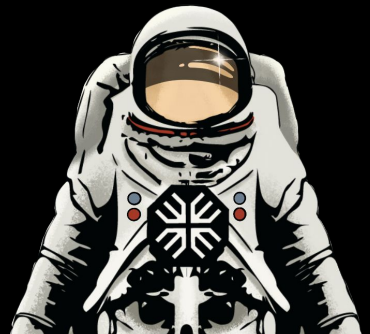
MAXIO

Dziękuję!



MAXIO

Dziękuję!



MAXIO

Avro

Avro is an open-source data serialization system. Part of the Apache ekosystem.

- defined by a schema
- data is fully typed
- stores data in JSON or binary format
- allows data serialization and deserialization
- supports multiple programming languages

MAXIO